Date=12/10/2020
Lecture By=Manish Mahant
Subject ⇒Revision Classes

In Today's Lecture

Functions in Javascript

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output. To use a function, you must define it somewhere in the scope from which you wish to call it.

Defining functions

A **function definition** (also called a **function declaration**, or **function statement**) consists of the `function` keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, `{...}`.

For example, the following code defines a simple function named `square`:

```
function square(number) {

return number * number;

}
```

The function `square` takes one parameter, called `number`. The function consists of one statement that says to return the parameter of the function (that is, `number`) multiplied by itself. The statement `return` specifies the value returned by the function:

```
return number * number;
```

Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

If you pass an object (i.e. a non-primitive value, such as `Array` or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
    theObject.make = 'Toyota';
  }

  var mycar = {make: 'Honda', model: 'Accord', year: 1998};
  var x, y;

  x = mycar.make; // x gets the value "Honda"
```

```
  myFunc(mycar);
  y = mycar.make; // y gets the value "Toyota"
                  // (the make property was changed by the function)
```

Function expressions
While the function declaration above is syntactically a statement, functions can also be created by a function expression.

Such a function can be **anonymous**; it does not have to have a name. For example, the function `square` could have been defined as:

```
const square = function(number) { return number * number }
var x = square(4) // x gets the value 16
```

However, a name *can* be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) }

console.log(factorial(3))
```

In the following code, the function receives a function defined by a function expression and executes it for every element of the array received as a second argument.

```
function map(f, a) {
    let result = []; // Create a new Array
    let i; // Declare variable
    for (i = 0; i != a.length; i++)
      result[i] = f(a[i]);
    return result;
  }
  const f = function(x) {
     return x * x * x;
  }
  let numbers = [0, 1, 2, 5, 10];
  let cube = map(f,numbers);
  console.log(cube);
```

Calling functions

Defining a function does not *execute* it. Defining it simply names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function `square`, you could call it as follows:

```
square(5);
```

The preceding statement calls the function with an argument of `5`. The function executes its statements and returns the value `25`.

Functions must be *in scope* when they are called, but the function declaration can be hoisted (appear below the call in the code), as in this example:

```
console.log(square(5));
/* ... */
function square(n) { return n * n }
```

Note: This works only when defining the function using the above syntax (i.e. `function funcName() {}`). The code below will not work.

This means that function hoisting only works with function *declarations*—not with function *expressions*.

```
console.log(square)    // square is hoisted with an initial value
undefined.
console.log(square(5)) // Uncaught TypeError: square is not a function
const square = function(n) {
  return n * n;
}
```

Javascript Objects

Objects in JavaScript, just as in many other programming languages, can be compared to objects in real life. The concept of objects in JavaScript can be understood with real life, tangible objects.

In JavaScript, an object is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design,

weight, a material it is made of, etc. The same way, JavaScript objects can have properties, which define their characteristics.

Objects and properties

A JavaScript object has properties associated with it. A property of an object can be explained as a variable that is attached to the object. Object properties are basically the same as ordinary JavaScript variables, except for the attachment to objects. The properties of an object define the characteristics of the object. You access the properties of an object with a simple dot-notation

```
objectName.propertyName
```

Like all JavaScript variables, both the object name (which could be a normal variable) and property name are case sensitive. You can define a property by assigning it a value. For example, let's create an object named `myCar` and give it properties named `make`, `model`, and `year` as follows:

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```

The above example could also be written using an **object initializer**, which is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`):

```
var myCar = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
```

Unassigned properties of an object are `undefined` (and not `null`).

```
myCar['make'] = 'Ford';
myCar['model'] = 'Mustang';
myCar['year'] = 1969;
```

An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation. This notation is also very useful when property names are to be dynamically determined (when the property name is not determined until runtime). Examples are as follows:

```
// four variables are created and assigned in a single go,
// separated by commas
var myObj = new Object(),
    str = 'myString',
    rand = Math.random(),
    obj = new Object();

myObj.type              = 'Dot syntax';
myObj['date created']   = 'String with space';
myObj[str]              = 'String value';
myObj[rand]             = 'Random Number';
myObj[obj]              = 'Object';
myObj['']               = 'Even an empty string';

console.log(myObj);
```

Using `this` for object references

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have 2 objects, `Manager`and `Intern`. Each object have their own `name`, `age` and `job`. In the function `sayHi()`, notice there is `this.name`. When added to the 2 objects they can be called and returns the `'Hello, My name is'` then adds the `name` value from that specific object. As shown below.

```
const Manager = {
    name: "John",
```

```
    age: 27,
    job: "Software Engineer"
  }
  const Intern= {
    name: "Ben",
    age: 21,
    job: "Software Engineer Intern"
  }

  function sayHi() {
      console.log('Hello, my name is', this.name)
  }

  // add sayHi function to both objects
  Manager.sayHi = sayHi;
  Intern.sayHi = sayHi;

  Manager.sayHi() // Hello, my name is John'
  Intern.sayHi() // Hello, my name is Ben'
```

The `this` refers to the object that it is in. You can create a new function called `howOldAmI()` which logs a sentence saying how old the person is.

```
function howOldAmI (){
    console.log('I am ' + this.age + ' years old.')
  }
  Manager.howOldAmI = howOldAmI;
  Manager.howOldAmI() // I am 27 years old.
```

Arrays in Javascript

Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non-contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.

Common operations

Create an Array

```
let fruits = ['Apple', 'Banana']

console.log(fruits.length)
// 2
```

Access an Array item using the index position

```
let first = fruits[0]
// Apple

let last = fruits[fruits.length - 1]
// Banana
```

Loop over an Array

```
fruits.forEach(function(item, index, array) {
    console.log(item, index)
  })
  // Apple 0
  // Banana 1
```

To know more About This
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Questions For Self Practice

https://au-assignment.s3.ap-south-1.amazonaws.com/Week_1_Day_1_Assignment-3035f7d6-b9f7-4283-b29d-296766c8d39d.pdf

https://au-assignment.s3.ap-south-1.amazonaws.com/Week_1_Day_1_Challenge-30daca70-9d1a-4d38-8e4d-3d0eb4f988c4.pdf